django-gisserver

Release 1.2.3

Jan 11, 2023

Usage Guide:

Django speaking WFS 2.0 to expose geo data.

CHAPTER 1

Features

- WFS 2.0 Basic implementation.
- GML 3.2 output.
- Standard and spatial filtering (FES 2.0)
- GeoJSON and CSV export formats.
- Extensible view/operations.
- Uses GeoDjango queries for filtering.
- Streaming responses for large datasets.

1.1 Getting Started

The django-gisserver module is designed to be used in an existing GeoDjango project. Hence, all configuration is done in code.

Install the module in your project:

```
pip install django-gisserver
```

Add it to the `INSTALLED_APPS`:

```
INSTALLED_APPS = [
    ...
    "gisserver",
]
```

Suppose the project has this exisiting GeoDjango model:

```
from django.contrib.gis.db.models import PointField
from django.db import models
```

(continues on next page)

(continued from previous page)

```
class Restaurant(models.Model):
    name = models.CharField(max_length=200)
    location = PointField(null=True)

    def __str__(self):
        return self.name
```

... then, the WFS logic can be exposed by writing a view.

```
from gisserver.features import FeatureType, ServiceDescription
from gisserver.geometries import CRS, WGS84
from gisserver.views import WFSView
from .models import Restaurant

RD_NEW = CRS.from_srid(28992)


class PlacesWFSView(WFSView):
    """An simple view that uses the WFSView against our test model."""

    xml_namespace = "http://example.org/gisserver"

    # The service metadata
    service_description = ServiceDescription(
        title="Places",
        abstract="Unittesting",
        keywords=["django-gisserver"],
        provider_name="Django",
        provider_site="https://www.example.com/",
        contact_person="django-gisserver",
    )

    # Each Django model is listed here as a feature.
    feature_types = [
        FeatureType(
            Restaurant.objects.all(),
            fields="__all__",
            other_crs=[RD_NEW]
        ),
    ]
```

Note: The list of `feature_types` lists all models that are exposed by this single view. Typically, a WFS server exposes a collection of related features on a single endpoint.

Use that view in the URLConf:

```
from django.urls import path
from . import views

urlpatterns = [
    path("/wfs/places/", views.PlacesWFSView.as_view()),
]
```

1.1.1 Testing the Server

You can now use http://localhost:8000/wfs/places/ in your GIS application. It will perform requests such as:

- http://localhost:8000/wfs/places/?SERVICE=WFS&REQUEST=GetCapabilities&ACCEPTVERSIONS=2.0.0,1.1.0,1.0.0
- http://localhost:8000/wfs/places/?SERVICE=WFS&REQUEST=DescribeFeatureType&VERSION=2.0.0&TYPENAMES=restaurant
- http://localhost:8000/wfs/places/?SERVICE=WFS&REQUEST=GetFeature&VERSION=2.0.0&TYPENAMES=restaurant&STARTINDEX=0&COUNT=1000&SRSNAME=urn:ogc:def:crs:EPSG::28992

1.1.2 Specifying the Output Format

By adding `&OUTPUTFORMAT=geojson` or `&OUTPUTFORMAT=csv` to the `GetFeature` request, the GeoJSON and CSV outputs are returned. These formats have an unlimited page size by default, as they're quite efficient.

1.2 Feature Type Configuration

Having completed the *getting started* page, a server should be running. The exposed feature types can be configured further.

Tip: WFS uses the term "feature" reference any real-world pointable thing, which is typically called an "object instance" in Django terminology. Likewise, a "feature type" describes the definition, which Django calls a "model".

1.2.1 Defining the Exposed Fields

By default, only the geometry field is exposed as WFS attribute. This avoids exposing any privacy sensitive fields.

While `fields="__all__"` works for convenience, it's better and more secure to define the exact field names using the `FeatureType(..., fields=[...])` parameter:

```
from gisserver.features import FeatureType
from gisserver.views import WFSView


class CustomWFSView(WFSView):
    ...

    feature_types = [
        FeatureType(
            Restaurant.objects.all(),
            fields=[
                "id",
                "name",
                "location",
                "owner_id",
                "created"
            ],
        ),
    ]
```

Renaming Fields

Using the `model_attribute`, the field name can differ from the actual attribute:

```
from gisserver.features import FeatureType, field
from gisserver.views import WFSView


class CustomWFSView(WFSView):
    ...

    feature_types = [
        FeatureType(
            Restaurant.objects.all(),
            fields=[
                "id",
                "name",
                field("location", model_attribute="point"),
                field("owner.id", model_attribute="owner_id"),
                "created"
            ],
        ),
    ]
```

Exposing Complex Fields

Foreign key relations can be exposed as "complex fields":

```
from gisserver.features import FeatureType, field
from gisserver.views import WFSView


class CustomWFSView(WFSView):
    ...

    feature_types = [
        FeatureType(
            Restaurant.objects.all(),
            fields=[
                "id",
                "name",
                "location",
                field("owner", fields=["id", "name", "phonenumber"])
                "created"
            ],
        ),
    ]
```

These fields appear as nested properties in the `GetFeature` response.

Exposing Flattened Relations

Since various clients (like QGis) don't support complex types well, relations can also be flattened by defining dotted-names. This can be combined with `model_attribute` which allows to access a different field:

```
from gisserver.features import FeatureType, field
from gisserver.views import WFSView


class CustomWFSView(WFSView):
    ...

    feature_types = [
        FeatureType(
            Restaurant.objects.all(),
            fields=[
                "id",
                "name",
                "location",
                field("owner.id", model_attribute="owner_id")
                "owner.name",
                field("owner.phone", model_attribute="owner.telephone"),
                "created"
            ],
        ),
    ]
```

If a dotted-name is found, the `field()` logic assumes it's a flattened relation.

In the example above, the `owner.id` field is linked to the `owner_id` model attribute so no additional JOIN is needed to filter against `owner.id`.

Overriding Value Retrieval

Changed in version 1.0.4: The `xsd_class` simplifies field overriding, and `value_from_object()` is now used.

Deep down, all feature fields are mapped to `XsdElement` objects that defines what WFS-server should generate as type definition. Field values are retrieved using `XsdElement.get_value()`, which resolves any dotted paths and calls Django's `value_from_object()` on the model field. This logic can be overwritten:

```
from gisserver.features import field
from gisserver.types import XsdElement
from gisserver.views import WFSView


class CustomXsdElement(XsdElement):
    def get_value(self, instance):
        return self.source.object_from_image(instance)


class CustomWFSView(WFSView):
    ...

    feature_types = [
        FeatureType(
            fields=[
               "id",
               "name",
               field("image", xsd_class=CustomXsdElement),
            ]
        )
    ]
```

1.3 Configuration Settings

The following configuration settings can be used to tweak server behavior.

The defaults are:

```
import math

# Flags
GISSERVER_CAPABILITIES_BOUNDING_BOX = True
GISSERVER_USE_DB_RENDERING = True
GISSERVER_SUPPORTED_CRS_ONLY = True

# Max page size
GISSERVER_DEFAULT_MAX_PAGE_SIZE = 5000
GISSERVER_GEOJSON_MAX_PAGE_SIZE = math.inf
GISSERVER_CSV_MAX_PAGE_SIZE = math.inf

# For debugging
GISSERVER_WRAP_FILTER_DB_ERRORS = True
GISSERVER_WFS_STRICT_STANDARD = False
```

1.3.1 GISSERVER_CAPABILITIES_BOUNDING_BOX

By default, the `GetCapabilities` response includes the bounding box of each feature. Since this is an expensive operation for large datasets, this can be disabled entirely.

If the project has the `CACHES` setting configured, the result will be briefly stored in a cache.

1.3.2 GISSERVER_USE_DB_RENDERING

By default, complex GML, GeoJSON and EWKT fragments are rendered by the database. This gives a better performance compared to GeoDjango, which needs to perform C-API calls indo GDAL for every coordinate of a geometry.

However, if you're not using PostgreSQL+PostGIS, you may want to disable this optimization.

1.3.3 GISSERVER_SUPPORTED_CRS_ONLY

By default, clients may only request features in one of the supported coordinate reference systems that the `FeatureType` has listed. Often databases (such as PostGIS) and the GDAL backend support a lot more out of the box. By disabling this setting, all system-wide supported CRS values can be used in the `?SRSNAME=...` parameter.

For performance reasons, the last 100 GDAL `CoordTransform` objects are stored in-memory. Allowing clients to change the output format so freely may cause some performance loss there.

1.3.4 GISSERVER_..._MAX_PAGE_SIZE

The `GISSERVER_*_MAX_PAGE_SIZE` settings allow to limit what the maximum requestable page size is. For GeoJSON and CSV, this is set to an infinite number which disables paging unless the `?COUNT=...` request parameter is used.

Note: QGis often requests 1000 features per request, regardless of the maximum page size. Custom `OutputRenderer` subclasses may also override this setting.

1.3.5 GISSERVER_WFS_STRICT_STANDARD

By default, the server is configured to pass CITE conformance tests. Strictly speaking, the WFS server should return an exception when an invalid `RESOURCEID` format is offered that doens't follow the "typename.identifier" notation.

1.3.6 GISSERVER_WRAP_FILTER_DB_ERRORS

By default, filter errors are nicely wrapped inside a WFS exception. This can be disabled for debugging purposes.

1.4 Troubleshooting

While most errors should be self-explanatory, this page lists anything that might be puzzling.

1.4.1 Operation on mixed SRID geometries

The error "Operation on mixed SRID geometries" often indicates that the database table uses a different SRID then the `GeometryField(srid=..)` configuration in Django assumes.

1.4.2 Only numeric values of degree units are allowed on geographic DWithin queries

The `DWithin` / `Beyond` can only use unit-based distances when the model field defines a projected system (e.g. `PointField(srid=...)`). Otherwise, only the units of the geometry field are supported (e.g. degrees for WGS84). If it's possible to work around this limitation, a pull request is welcome.

1.4.3 ProgrammingError / InternalError database exceptions

When an `ProgrammingError` or `InternalError` happens, this likely means the database table schema doesn't match with the Django model. As WFS queries allow clients to construct complex queries against a table, any discrepancies between the Django model and database table are bound to show up.

For example, if your database table uses an `INTEGER` or `CHAR(1)` type, but declares a `BooleanField` in Django this will cause errors. Django can only construct queries in reliably when the database schema matches the model definition.

Make sure your Django model migrations have been applied, or that any imported database tables matches the model definition.

1.4.4 InvalidCursorName cursor "_django_curs_..." does not exist

This error happens when the database connection passes through a connection pooler (e.g. PgBouncer). One workaround is wrapping the view inside `@transaction.atomic`, or disabling server-side cursors entirely by adding `DISABLE_SERVER_SIDE_CURSORS = True` to the settings.

For details, see: https://docs.djangoproject.com/en/stable/ref/databases/#transaction-pooling-server-side-cursors

1.4.5 Sentry SDK truncates the exceptions for filters

The Sentry SDK truncates log messages after 512 characters. This typically truncates the contents of the `FILTER` parameter, as it's XML notation is quite verbose. Add the following to your settings file to see the complete message:

```
import sentry_sdk.utils

sentry_sdk.utils.MAX_STRING_LENGTH = 2048  # for WFS FILTER exceptions
```

1.5 Overriding Server Logic

There are a few places where the server logic can be extended:

There are a few places that allow to customize the WFS logic:

1.5.1 View Layer

The following methods of the `WFSView` can be overwritten:

- `get_feature_types()` to dynamically generate all exposed features.
- `get_service_description()` to dynamically generate the description.
- `dispatch()` to implement basic auth.

1.5.2 Feature Layer

Overriding `FeatureType` allows to change how particular features and fields are exposed. It can also override the internal XML Schema Definition (XSD) that all output and query filters read.

This can also adjust the

- Overriding `check_permissions()` allows to perform a permission check before the feature can be read (e.g. a login role check).
- Overriding `get_queryset()` allows to define the queryset per request.
- Overriding `xsd_type` constructs the internal XSD definition of this feature.
- Overriding `xsd_type_class` defines which class constructs the XSD.

The `field()` function returns a `FeatureField`. Instances of this class can be passed directly to the `FeatureType(fields=...)` parameter, and override these attributes:

- `xsd_element` constructs the internal XSD that filters and output formats use.
- `xsd_element_class` defines which class defines the attribute.

1.5.3 XSD Layer

The feature fields generate an internal XML Schema Definition (XSD) that defines how properties are read, and where the underlying ORM field/relation can be found. These types can be overwritten for custom behavior, and then be returned by custom `FeatureType` and `FeatureField` objects.

- `XsdComplexType` defines a complete class with elements and attributes.
- `XsdElement` defines a property that becomes a normal element.
- `XsdAttribute` defines the attributes (only `gml:id` is currently rendered).

The elements and attributes have the following fields:

- `orm_path` - returns where to find the ORM relation.
- `orm_field` - returns the first part of the ORM relation.
- `orm_relation` - returns the ORM relation as path and final field name.
- `get_value()` - how to read the attribute value.
- `format_value()` - format raw-retrieved values from the database (e.g `.values()` query).
- `to_python()` - how to cast input data.
- `validate_comparison()` - checks a field supports a certain data type.
- `build_lhs_part()` - how to generate the ORM left-hand-side.
- `build_rhs_part()` - how to generate the ORM right-hand-side.

1.5.4 Custom Filter Functions

> **Warning:** While the machinery to hook new functions is in place, this part is still in development.

As part of the WFS Filter Encoding, a client can execute a function against a server. These are executed with `?REQUEST=GetFeature&FILTER...`

An expression such as: **table_count == Add("previous_table_count", 100)** would be encoded in the following way using the Filter Encoding Specification (FES):

```
<fes:Filter xmlns:fes="http://www.opengis.net/fes/2.0">
    <fes:PropertyIsEqualTo>
        <fes:ValueReference>table_count</fes:ValueReference>
        <fes:Function name="Add">
            <fes:ValueReference>previous_table_count</fes:ValueReference>
            <fes:Literal>100</fes:Literal>
        </fes:Function>
    </fes:PropertyIsEqualTo>
</fes:Filter>
```

These FES functions can be defined in the project, by generating a corresponding database function.

Use `gisserver.parsers.fes_function_registry` to register new functions:

```
from django.db.models import functions
from gisserver.parsers import fes_function_registry
from gisserver.types import XsdTypes
```

(continues on next page)

(continued from previous page)

```
# Either link an exising Django ORM function:

function_registry.register(
    "atan",
    functions.ATan,
    arguments={"value": XsdTypes.double},
    returns=XsdTypes.double,
)


# Or link a parsing logic that generates an ORM function/object:

@function_registry.register(
    name="Add",
    arguments=dict(value1=XsdTypes.double, value2=XsdTypes.double),
    returns=XsdTypes.double,
)
def fes_add(value1, value2):
    return F(value1) + value2
```

Each FES function should return a Django ORM `Func` or `Combinable` object.

1.5.5 Custom Stored Procedures

Warning: While the machinery to add new stored procedures is in place, this part is still in development.

Aside from filters, a WFS server can also expose "stored procedures". These are executed with `?REQUEST=GetFeature&STOREDQUERY_ID=...` By default, only `GetFeatureById` is built-in.

These stored procedures can be defined like this:

```
from gisserver.queries import StoredQuery, stored_query_registry
from gisserver.types import XsdTypes


@stored_query_registry.register(
    id="GetRecentChanges",
    title="Get all recent changed features",
    abstract="All recent changes...",
    parameters={"date": XsdTypes.date},
)
class GetRecentChanges(StoredQuery):
    ...
```

For a simple implementation, the following methods need to be overwritten:

- `get_type_names()` defines which feature types this query applies to.
- `compile_query()` defines how to filter the queryset.

For full control, these methods can also be overwritten instead:

- `get_queryset()` to define the full results.

- `get_hits()` to return the collection for `RESULTTYPE=hits`.
- `get_results()` to return the collection for `RESULTTYPE=results`.

1.6 WFS User Manual

This is a brief explanation of using a WFS server.

Commonly, a WFS server can is accessed by GIS-software, such as QGis. The URL that's configured inside `urls.py` can be used directly as WFS endpoint. For example, add https://api.data.amsterdam.nl/v1/wfs/gebieden/ to QGis.

Everything, for querying and viewing can be done in QGis.

Tip: The parameters `?SERVICE=WFS&VERSION=2.0.0&REQUEST=..` are appended to the URL by QGis. It's not required to add these yourself.

The WFS server can also be accessed directly from a HTTP client (e.g. curl) or web browser. In such case, use the basic URL above, and include the query parameters:

`?SERVICE=WFS&VERSION=2.0.0&REQUEST=GetFeature&TYPENAMES=`*`featurename`*

The available feature types can be found in the **GetCapabilities** request:

`?SERVICE=WFS&VERSION=2.0.0&REQUEST=GetCapabilities`

The remaining part of this page assumes this manual access.

1.6.1 Export Formats

The following export formats are available:

- GeoJSON
- CSV

These can be queried by manually crafting a **GetFeature** request. The parameters `TYPENAMES=`*`feature-name`* and `OUTPUTFORMAT=`*`format`* should be included.

For example:

- ...&TYPENAMES=wijken&OUTPUTFORMAT=geojson
- ...&TYPENAMES=wijken&OUTPUTFORMAT=csv

Tip: In the example links above, a `COUNT=` parameter is included to activate pagination. When this parameter is omitted, *all objects* will be returned in a single request. For most datasets, the server is capable of efficiently delivering all results in a single response.

1.6.2 Geometry Projections

The exportlink can be extended with the `SRSNAME` parameter to define the geometry projection of all geo data. For example, `SRSNAME=urn:ogc:def:crs:EPSG::3857` activates the web-mercator projection which is used by Google Maps. A common default is `urn:ogc:def:crs:EPSG::4326`, which is the worldwide WGS 84 longitude-latitude.

1.6.3 Simple Filters

The WFS protocol offers a powerful syntax to filter data. Use the request `REQUEST=GetFeature` with a `FILTER` argument. The filter value is expressed as XML.

For example, to query all neighbourhoods (typename buurten) of the central district (stadsdeel) in Amsterdam:

```
<Filter>
    <PropertyIsEqualTo>
        <ValueReference>ligt_in_stadsdeel/naam</ValueReference>
        <Literal>Centrum</Literal>
    </PropertyIsEqualTo>
</Filter>
```

This can be included in the request, for example:

- ...&TYPENAMES=wijken&OUTPUTFORMAT=geojson&FILTER=%3CFilter%3E%3CPropertyIsEqualTo%3E%3CValueRef

The `FILTER` parameter replaces the separate `BBOX` and `RESOURCEID` parameters. If you use these parameters as well, they should be included in the filter:

```
<Filter>
    <And>
        <BBOX>
            <gml:Envelope srsName="EPSG:4326">
                <gml:lowerCorner>4.58565 52.03560</gml:lowerCorner>
                <gml:upperCorner>5.31360 52.48769</gml:upperCorner>
            </gml:Envelope>
        </BBOX>
        <PropertyIsEqualTo>
            <ValueReference>status</ValueReference>
            <Literal>1</Literal>
        </PropertyIsEqualTo>
    </And>
</Filter>
```

The `RESOURCEID` parameter has a `<ResourceId>` equivalent which can appear several times in the filter:

```
<Filter>
    <ResourceId rid="TYPENAME.123" />
    <ResourceId rid="TYPENAME.4325" />
    <ResourceId rid="OTHERTYPE.567" />
</Filter>
```

1.6.4 Complex Filters

The WFS Filter Encoding Standaard (FES) supports many operators. These tags are all supported:

Element	SQL equivalent	Description
<PropertyIsEqualTo>	*a* == *b*	Values must be equal.
<PropertyIsNotEqualTo>	*a* != *b*	Values must not be equal.
<PropertyIsLessThan>	*a* < *b*	Value 1 must be less than value 2.
<PropertyIsGreaterThan>	*a* > *b*	Value 1 must be greater than value 2.
<PropertyIsLessThanOrEqualTo>	*a* <= *b*	Value 1 must be less than or equal to value 2.
<PropertyIsGreaterThanOrEqualTo>	*a* >= *b*	Value 1 must be greater than or equal to value 2.
<PropertyIsBetween>	*a* BETWEEN *x* AND *y*	Compares between <LowerBoundary> and <UpperBoundary>, which both contain an expression.
<PropertyIsLike>	*a* LIKE *b*	Performs a wildcard comparison.
<PropertyIsNil>	*a* IS NULL	Value must be NULL (xsi:nil="true" in XML).
<PropertyIsNull>	n.a.	Property may not exist (currently implemented as <PropertyIsNil>).
<BBOX>	ST_Intersects(*a*, *b*)	Geometry must be in value 2. The field name may be omitted to use the default.
<Contains>	ST_Contains(*a*, *b*)	Geometry 1 completely contains geometry 2.
<Crosses>	ST_Crosses(*a*, *b*)	The geometries have some common interior points.
<Disjoint>	ST_Disjoint(*a*, *b*)	The geometries are not connected in any way.
<Equals>	ST_Equals(*a*, *b*)	The geometries are identical.
<Intersects>	ST_Intersects(*a*, *b*)	The geometries share some space.
<Touches>	ST_Touches(*a*, *b*)	The edges of the geometries touch each other.
<Overlaps>	ST_Overlaps(*a*, *b*)	The geometries overlap.
<Within>	ST_Within(*a*, *b*)	Geometry 1 is completely contained within geometry 2.
<DWithin>	ST_DWithin(*a*, *b*, *d*)	The geometries are within a given distance of each other.
<Beyond>	NOT ST_DWithin(*a*, *b*, *d*)	The geometries are not within a given distance.
<And>	*a* AND *b*	The nested elements must all be true.
<Or>	*a* OR *b*	Only one of the nested elements has to be true.
<Not>	NOT *a*	Negation of the nested element.
<ResourceId>	table.id == *value* / table.id IN (*v1*, *v2*, ...)	Searches only one element for "type name.identifier". Combines multiple elements into an IN query.

Tip: For the <BBOX> operator the geometry field may be omitted. The standard geometry field is then used (usually the first field).

Note: Although a number of geometry operators seem to be identical for surfaces (such as <Intersects>, <Crosses> and <Overlaps>), their mutual differences are particularly visible when comparing points with surfaces.

Various expressions may be used as values:

Expression	SQL equivalent	Description
`<ValueReference>`	*field-name*	References a field.
`<Literal>`	value	Literal value, can also be a GML-object.
`<Function>`	*function-name*`(..)`	Executes a function, such as `abs`, `sin`, `strLength`.
`<Add>`	*a* `+` *b*	Addition (WFS 1 expression).
`<Sub>`	*a* `-` *b*	Subtraction (WFS 1 expression).
`<Mul>`	*a* `*` *b*	Multiplication (WFS 1 expression).
`<Div>`	*a* `/` *b*	Division (WFS 1 expression).

This allows to create complex filters, such as:

```
<Filter>
    <And>
        <PropertyIsEqualTo>
            <ValueReference>status</ValueReference>
            <Literal>1</Literal>
        </PropertyIsEqualTo>
        <Or>
            <PropertyIsEqualTo>
                <ValueReference>container_type</ValueReference>
                <Literal>Other</Literal>
            </PropertyIsEqualTo>
            <PropertyIsEqualTo>
                <ValueReference>container_type</ValueReference>
                <Literal>Textile</Literal>
            </PropertyIsEqualTo>
            <PropertyIsEqualTo>
                <ValueReference>container_type</ValueReference>
                <Literal>Glass</Literal>
            </PropertyIsEqualTo>
            <PropertyIsEqualTo>
                <ValueReference>container_type</ValueReference>
                <Literal>Papier</Literal>
            </PropertyIsEqualTo>
            <PropertyIsEqualTo>
                <ValueReference>container_type</ValueReference>
                <Literal>Organic</Literal>
            </PropertyIsEqualTo>
            <PropertyIsEqualTo>
                <ValueReference>container_type</ValueReference>
                <Literal>Plastic</Literal>
            </PropertyIsEqualTo>
        </Or>
    </And>
</Filter>
```

1.6.5 Functions

Functions are executed by using the tag `<Function name="..">..</Function>`. This can be used anywhere as an expression instead of a `<ValueReference>` or `<Literal>`.

Inside the function, the parameters are also given as expressions: a `<ValueReference>`, `<Literal>` or new `<Function>`. As a simple example:

```
<fes:Function name="sin">
    <fes:ValueReference>fieldname</fes:ValueReference>
</fes:Function>
```

The following functions are available in the server:

Functie	SQL equivalent	Description
`strConcat(string)`	`CONCAT()`	Concatenates strings
`strToLowerCase(string)`	`LOWER()`	Convert text to lowercase.
`strToUpperCase(string)`	`UPPER()`	Convert text to uppercase.
`strTrim(string)`	`TRIM()`	Remove white space at the beginning and e
`strLength(string)`	`LENGTH()` / `CHAR_LENGTH()`	Determines text length.
`length(string)`	`LENGTH()` / `CHAR_LENGTH()`	Alias of `strLength()`.
`abs(number)`	`ABS()`	Invert negative numbers.
`ceil(number)`	`CEIL()`	Rounding up.
`floor(number)`	`FLOOR()`	Rounding down.
`round(value)`	`ROUND()`	Regular rounding.
`min(value1, value2)`	`LEAST()`	Uses the smallest number.
`max(value1, value2)`	`GREATEST()`	Uses the largest number.
`pow(base, exponent)`	`POWER()`	Exponentiation
`exp(value)`	`EXP()`	Exponent of (2,71828...; natural logarithn
`log(value)`	`LOG()`	Logarithm; inverse of an exponent.
`sqrt(value)`	`SQRT()`	Square root, inverse of exponentiation.
`acos(value)`	`ACOS()`	Arccosine; inverse of cosine.
`asin(value)`	`ASIN()`	Arcsine; inverse van sine.
`atan(value)`	`ATAN()`	Arctangent; inverse of tangent.
`atan2(x, y)`	`ATAN2()`	Arctangent, for usage outside the range of a
`cos(radians)`	`COS()`	Cosine
`sin(radians)`	`SIN()`	Sine
`tan(radians)`	`TAN()`	Tangent
`pi()`	`PI`	The value of π (3,141592653...)
`toDegrees(radians)`	`DEGREES()`	Conversion of radians to degrees.
`toRadians(degree)`	`RADIANS()`	Conversion degrees to radians.
`Area(geometry)`	`ST_AREA()`	Convert geometry to area.
`Centroid(features)`	`ST_Centroid()`	Return geometric center as “gravity point”.
`Difference(geometry1, geometry2)`	`ST_Difference()`	Parts of geometry 1 that do not overlap witl
`distance(geometry1, geometry2)`	`ST_Distance()`	Minimum distance between 2 geometries.
`Envelope(geometry)`	`ST_Envelope()`	Convert geometry to bounding box.
`Intersection(geometry1, geometry2)`	`ST_Intersection()`	Parts of geometry 1 that overlap with geom
`Union(geometry1, geometry2)`	`ST_Union()`	Merge Geometry 1 and 2.

1.6.6 Filter Compatibility

Strictly speaking, XML namespaces are required in the filter. Since many clients omit them, the server also supports requests without namespaces. For the sake of completeness, a request with namespaces included looks like this:

```
<fes:Filter xmlns:fes="http://www.opengis.net/fes/2.0"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://www.opengis.net/fes/2.0
        http://schemas.opengis.net/filter/2.0/filterAll.xsd">
    <fes:PropertyIsEqualTo>
```

(continues on next page)

(continued from previous page)

```
        <fes:ValueReference>stadsdeel/naam</fes:ValueReference>
        <fes:Literal>Centrum</fes:Literal>
    </fes:PropertyIsEqualTo>
</fes:Filter>
```

When a geometry filter is included, this also requires the GML namespace:

```
<fes:Filter
    xmlns:fes="http://www.opengis.net/fes/2.0"
    xmlns:gml="http://www.opengis.net/gml/3.2"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.opengis.net/fes/2.0
    http://schemas.opengis.net/filter/2.0/filterAll.xsd
    http://www.opengis.net/gml/3.2 http://schemas.opengis.net/gml/3.2.1/gml.xsd">
    <fes:BBOX>
        <gml:Polygon gml:id="P1" srsName="http://www.opengis.net/def/crs/epsg/0/4326">
            <gml:exterior>
                <gml:LinearRing>
                    <gml:posList>10 10 20 20 30 30 40 40 10 10</gml:posList>
                </gml:LinearRing>
            </gml:exterior>
        </gml:Polygon>
    </fes:BBOX>
</fes:Filter>
```

According to the XML rules, the "fes" namespace alias can be renamed here or omitted if only `xmlns="..."` is used instead of `xmlns:fes="..."`.

Several existing clients still use other WFS 1 elements, such as `<PropertyName>` instead of of `<ValueReference>`. For compatibility this tag is also supported.

The WFS 1 expressions `<Add>`, `<Sub>`, `<Mul>` and `<Div>` are also implemented to support arithmetic operations from QGis (addition, subtraction, multiplication and division).

1.7 Standards Compliance

Some facts about this project:

- All operations for the WFS Basic conformance class are implemented.
- The CITE Test Suite only reveals a few bits left to implement.
- You should be able to view the WFS server QGis.
- The unit tests validate the output against WFS 2.0 XSD schema.

1.7.1 Unimplemented Bits

Some remaining parts for the "WFS simple" conformance level are not implemented yet:

- KVP filters: `propertyName`, `aliases`.
- Remote resolving: `resolveDepth`, `resolveTimeout`.
- Multiple queries in a single GET call.
- Some `GetCapabilities` features: `acceptFormats` and `sections`.

- Temporal filtering (high on todo)
- Tests on axis orientation.

1.7.2 Planned

- WFS-T (Transactional) support, which also needs HTTP POST requests.

1.7.3 Hopefully

While WMS and WMTS are not on the roadmap, they could be implemented based on Mapnik. Other Python tiling logic such as TileCache and TileStache could serve as inspiration too.

1.7.4 Low-Prio Items

Anything outside WFS-T could be implemented, but is very low on the todo-list:

- The methods for the WFS locking and inheritance conformance classes.
- SOAP requests.
- Other OGS protocols such as WCS
- Other output formats (shapefile, KML, GML 3.1) - but easy to add.

Some parts (such as output formats or missing WFS methods) can even be implemented within your own project, by overriding the existing class attributes.

1.7.5 Compatibility with older WFS-clients

Some popular WFS-clients still use aspects of the WFS 1.0 filter syntax in their queries. To support these clients, the following logic is also implemented:

- The `<PropertyName>` tag instead of `<fes:ValueReference>`
- The `<fes:Add>`, `<fes:Sub>`, `<fes:Mul>` and `<fes:Div>` arithmetic operators, used by QGis.
- The `FILTER=<Filter>...</Filter>` parameter without an XML namespace declaration, typically seen in web-browser libraries.
- The `MAXFEATURES` parameter instead of `COUNT`.
- The `TYPENAME` parameter instead of `TYPENAMES` (used by the CITE test suite!).
- Using `A` and `D` as sort direction in `SORTBY` / `<fes:SortBy>` instead of `ASC` and `DESC`.

For CITE test suite compliance, `urn:ogc:def:query:OGC-WFS::GetFeatureById` query returns an HTTP 404 for an invalid resource ID format, even though the WFS 2 specification states it should return an `InvalidParameterValue`. Likewise, the `<ResourceId>` query returns an empty list instead of `InvalidParameterValue` for invalid resource ID formats. This behavior can be disabled with the `GISSERVER_WFS_STRICT_STANDARD` setting.

1.8 Development

When you follow the source of the `WFSView`, `WFSMethod` and `Parameter` classes, you'll find that it's written with extensibility in mind. Extra parameters and operations can easily be added there. You could even do that within your own projects and implementations.

A lot of the internal classes and object names are direct copies from the WFS spec. By following these type definitions, a lot of the logic and code structure follows naturally.

The `Makefile` gives you all you need to start working on the project. Typing `make` gives an overview of all possible shortcut commands.

1.8.1 Running tests

The `Makefile` has all options. Just typing `make` gives a list of all commands.

Using `make test`, and `make retest` should run the pytest suite.

A special `make docker-test` runs the tests as they would run within Travis-CI. This helps to debug any differences between coordinate transformations due to different PROJ.4 versions being installed.

1.8.2 Accessing the CITE tests

To perform CITE conformance testing against a server, use https://cite.opengeospatial.org/teamengine/.

- At the bottom of the page, there is a **Create an account** button.
- Create a new WFS 2.0 test session
- At the next page, enter the URL to the `GetCapabilities` document, e.g.:

```
http://example.org/v1/wfs/?VERSION=2.0.0&REQUEST=GetCapabilities
```

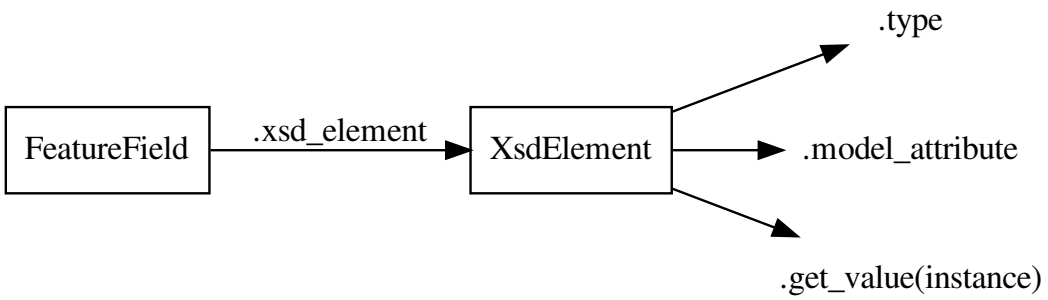
Local testing can't be done with NGrok, as it exceeds the rate limiting. Instead, consider opening a temporary port-forward at your router/modem.

1.8.3 Internal logic

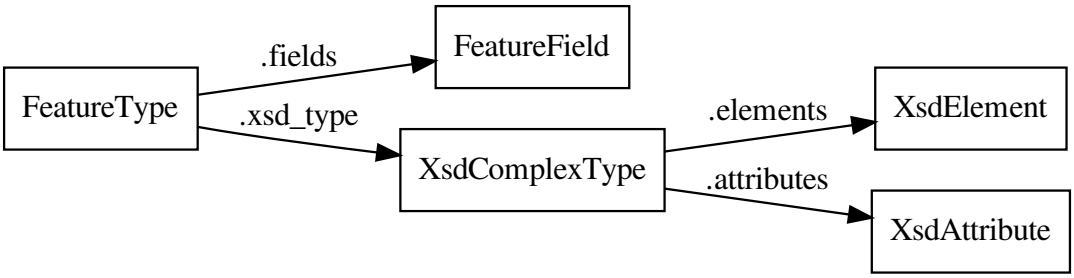
Features and Fields

Each `FeatureField` is transformed into an internal `XsdElement` object. The model field access happens through `XsdElement.get_value()`. Note that the `type` can either reference either an `XsdTypes` or

`XsdComplexType` object.



Each `FeatureType` is transformed into an internal `XsdComplexType` definition:



Data Retrieval

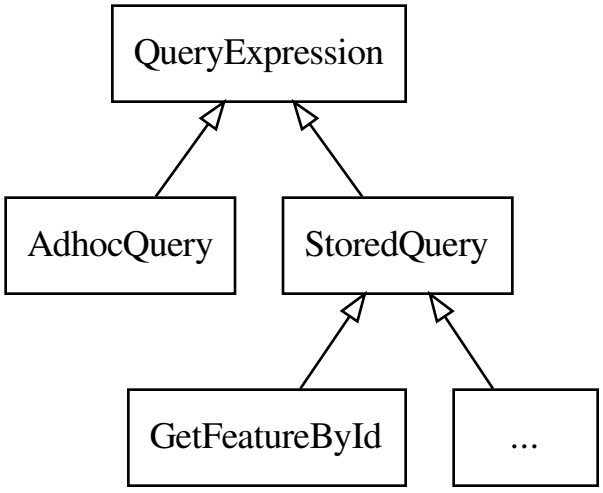
When `GetFeature` or `GetPropertyValue` is called, several things happen:

- Request parsing.
- Query construction.
- Query execution.
- Output rendering.

The whole `<fes:Filter>` contents is translated an an internal "abstract syntax tree" (AST) which closely resembles all class names that the FES standard defines.

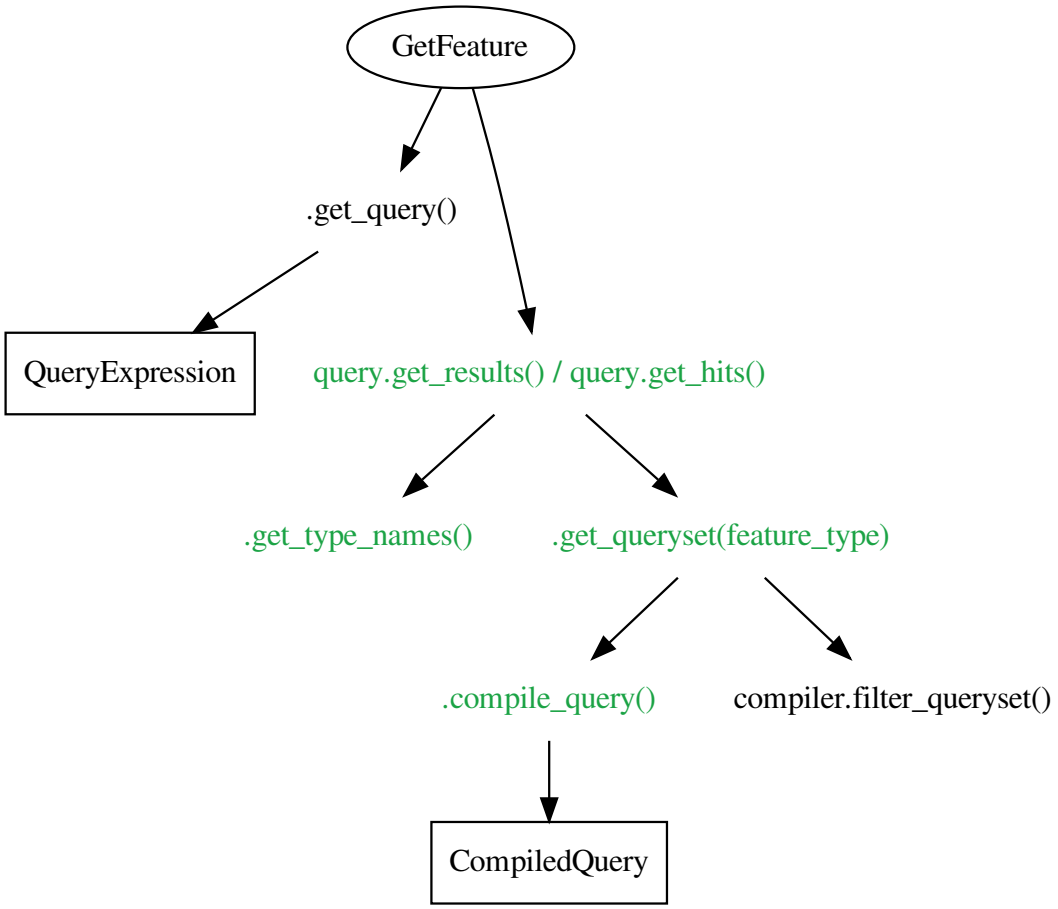
Then, the views `.get_query()` method constructs the proper query object based on the request parameters.

The query class diagram looks like:



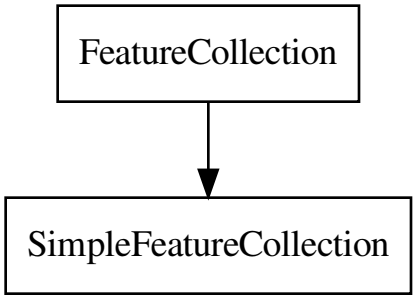
All regular requests such as `?FILTER=...`, `?BBOX=...`, `?SORTBY=...` and `?RESOURCEID=...` are handled by the `AdhocQuery` class. A subclass of `StoredQuery` is used for `?STOREDQUERY_ID=...` requests.

The query is executed:



The `CompiledQuery` collects all intermediate data needed to translate the `<fes:Filter>` queries to a Django ORM call. This object is passed though all nodes of the filter, so each `build...()` function can add their lookups and annotations.

Finally, the query returns a `FeatureCollection` that iterates over all results. Each `FeatureType` is represented by a `SimpleFeatureCollection` member.



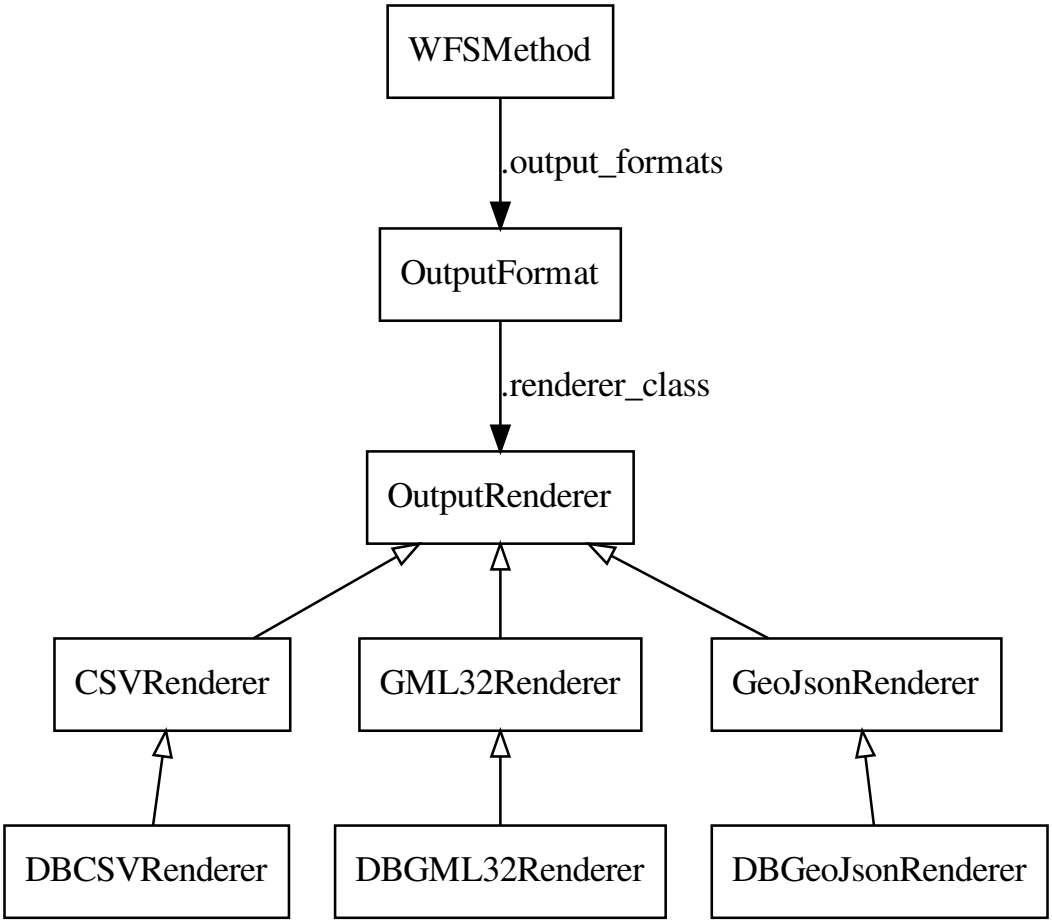
These collections attempt to use queryset-iterator logic as much as possible, unless it would cause multiple queries (such as needing the `number_matched` data early).

Output Rendering

Each `WFSMethod` has a list of `OutputFormat` objects:

```
class GetFeature(BaseWFSGetDataMethod):
    output_formats = [
        OutputFormat("application/gml+xml", version="3.2", renderer_class=output.
↪gml32_renderer),
        OutputFormat("text/xml", subtype="gml/3.2.1", renderer_class=output.gml32_
↪renderer),
        OutputFormat("application/json", subtype="geojson", charset="utf-8", renderer_
↪class=output.geojson_renderer),
        OutputFormat("text/csv", subtype="csv", charset="utf-8", renderer_
↪class=output.csv_renderer),
        # OutputFormat("shapezip"),
        # OutputFormat("application/zip"),
    ]
```

The `OutputFormat` class may reference an `renderer_class` which points to an `OutputRenderer` object.



Various output formats have an DB-optimized version where the heavy rendering of the EWKT, JSON or GML fragments is done by the database server. Most output formats return a streaming response for performance.

Alternatively, the `WFSMethod` may render an XML template using Django templates.

1.8.4 WFS Specification

The WFS specification and examples be found at:

- https://www.opengeospatial.org/standards/ (all OGC standards)
- https://docs.opengeospatial.org/ (HTML versions)

Some deeplinks:

- https://www.opengeospatial.org/standards/common (OGC Web Service Common)
- https://www.opengeospatial.org/standards/wfs#downloads (OGC WFS)
- https://portal.opengeospatial.org/files/09-025r2 (WFS 2.0 spec)

- https://portal.opengeospatial.org/files/09-026r1 (OpenGIS Filter Encoding 2.0)
- https://portal.opengeospatial.org/files/07-036 (GML 3.2.1)

Other links:

- http://schemas.opengis.net/wfs/2.0/ (XSD and examples)
- https://mapserver.org/development/rfc/ms-rfc-105.html (more examples)

Coordinate systems, and axis orientation:

- https://macwright.com/lonlat/ (the inconsistency of lat/lon or lon/lat)
- https://macwright.com/2015/03/23/geojson-second-bite.html (More than you ever wanted to know about GeoJSON)
- https://mapserver.org/ogc/wms_server.html#coordinate-systems-and-axis-orientation (mapserver WMS part)
- https://mapserver.org/ogc/wfs_server.html#axis-orientation-in-wfs-1-1-and-2-0 (mapserver WFS part)
- https://docs.geoserver.org/stable/en/user/services/wms/basics.html#axis-ordering (geoserver WMS part)
- https://docs.geoserver.org/stable/en/user/services/wfs/axis_order.html (geoserver WFS part)

CHAPTER 2

Why this code is shared

The “datapunt” team of the Municipality of Amsterdam develops software for the municipality. Much of this software is then published as Open Source so that other municipalities, organizations and citizens can use the software as a basis and inspiration to develop similar software themselves. The Municipality of Amsterdam considers it important that software developed with public money is also publicly available.

This package is initially developed by the City of Amsterdam, but the tools and concepts created in this project can be used in any city.